Framework based on design patterns for providing persistence in object-oriented programming languages

J. Kienzle and A. Romanovsky

Abstract: An approach is described providing object persistence in object-oriented programming languages without modifying the run-time system or the language itself. By successively applying design patterns such as the 'serialiser', 'factory method', and 'strategy' patterns we develop an object-oriented framework for providing object persistence. The advantages of object-orientation are highlighted: structured classification through class-hierarchies, extensibility and promotion of reuse. The framework clearly separates persistence control from storage control. A hierarchy of different storage types, useful in different application domains, is introduced. The framework does not rely on any kind of special programming language features. It only uses basic object-oriented programming techniques, and is therefore implementable in any object-oriented programming language. An experimental implementation in Ada 95 is presented.

1 Introduction

In recent years research into persistent programming languages and systems has shown that this technology is useful for developing complex software in many problem domains. Persistence is used whenever data values from a program execution are saved so that they can be used in a later execution. Software-fault-tolerance mechanisms based on backward error recovery use persistence to provide state restoration in case of computer crashes. Transaction durability is often achieved using persistence techniques. How the data is saved, and what kind of storage medium is used for that purpose depends on the applications demands and can vary considerably from one application to another. Unfortunately, widely used object-oriented programming languages still do not offer support for persistence.

2 Persistence and programming languages

In [1] persistence is defined as follows:

'Persistence is the property of an object through which its existence transcends time (i.e. the object continues to exist after its creator ceases to exist) and/or space (i.e. the object's location moves from the address space in which it was created)'.

There are many possible schemes for supporting persistence. For a complete survey refer to [2]. The most sophisticated and desired form of persistence is orthogonal persistence [3]. It is the provision of persistence for all data irrespective of their type. In a programming language providing orthogonal persistence, persistent data is created and used in the same way as nonpersistent data. Loading and saving of values does not alter their semantics, and the process is transparent to the application program. Whether or not data should be made persistent is often determined using a technique called persistence by reachability. The persistence support designates an object as a persistent root and provides applications with a built-in function for locating it. Any object that is 'reachable' from the persistent root, for instance by following pointers, is automatically made persistent.

Due to the demanding requirements of orthogonal persistence, implementations such as PS-Algol [4], Poly ML [5] or PJama [6] had to slightly modify the programming language and/or modify the run-time system. Oudshoorn and Crawley [7] investigate adding orthogonal persistence to the Ada 95 [8] language. The authors identified the following problems:

- Orthogonal persistence requires that both data and types can have indefinite lifetimes. If a persistent application is to evolve, structural equivalence and dynamic type checking are necessary when a program binds to an object from the persistent store. When introducing orthogonal persistence, type compatibility within an execution extends to type compatibility across different executions. This may conflict with the typing rules of the programming language.
- Often programming languages allow the use of static variables inside classes or even as standalone global variables. It is possible that a programmer uses such static variables to link objects together, for instance by using a static table that links key values to some other data. Now if the key values are made persistent, the table should also persist, or else the key values are useless. It might be tricky to provide automatic persistence for such static variables without breaking orthogonal persistence.

IEE Proceedings online no. 20020465
DOI: 10.1049/ip-sen:20020465
Paper first received 22nd August 2001 and in revised form 18th April 2002

J. Kienzle is with the Software Engineering Laboratory, Swiss Federal Institute of Technology, CH-1015 Lausanne Ecublens, Switzerland
A. Romanovsky is with the Department of Computing Science, University of Newcastle, Newcastle upon Tyne NE1 7RU, United Kingdom

such as task types/threads and subprogram pointer values persist. This can raise severe implementation problems.
- A program might evolve and change the definition of types and classes, but still try and work with values saved in previous executions. To make this work, some form of version control must be provided, and additional dynamic checking is required. The problem can be even more complicated when considering inheritance.
- Another important problem when providing orthogonal persistence is storage management. Persistent data that will not be used anymore must be deleted, for storage leaks will result in permanent loss of storage capacity. This basically requires some form of automatic garbage collection, at least for all persistent data.

Finally, the authors conclude that adding orthogonal persistence to the Ada 95 language would require major changes, making the new language backwards-incompatible. It is interesting that even in the case of the Java language, a modern object-oriented language that already provides automatic garbage collection and a powerful reflection mechanism, the virtual machine executing the Java byte code had to be modified to support orthogonal persistence [9].

If we do not strive for orthogonal persistence, there are multiple ways to provide persistence support for conventional programming languages. Many languages have been extended or provide standard libraries that allow data to be made persistent, for instance by saving it to disc. Avalon [10] for instance is an extension to C++ that provides persistence and transactions. The authors have extended the C++ language, providing an additional keyword 'stable', which is used to designate class attributes that are to be made persistent.

Persistence support in object-oriented programming languages must provide a mechanism that allows the state of an object to persist between different executions of an application. It can be quite challenging to find means for taking the in-memory representation of the objects state and writing it to some storage device. Fortunately, object-oriented programming languages often provide some kind of streaming functionality that allows transforming the state of an object into a flat stream of bytes. Some languages go even further and provide streams that allow a user to write objects into files or other storage devices (Ada `Stream_IO` (see A.12.1 of [8]), Java [11] `FileOutputStreams`). Unfortunately, the facilities provided by the programming language are not always sufficient, or they lack modularity and extensibility, making the definition of new persistent objects or the addition of new storage devices difficult or even impossible.

We believe that when designing persistence support for object-oriented programming languages one should strive to achieve the following:

- Clear separation of concerns: The persistent object should not know about storage devices or about the data format that is used when writing the state of the object onto the storage device and vice versa.
- Modularity and extensibility: It should be straightforward to define new persistent objects or add new storage devices.
- Safe storage management: Storage leaks leading to wasted space on the storage device must be prevented.

To help the programmer we propose a general framework for providing object persistence that maximises these goals. It can be used by two different types of programmers: persistence support programmers and application programmers. The persistence support programmers will add new storage devices to the framework using object-oriented programming techniques. Application programmers will use the framework to declare persistent objects. When instantiating a new persistent object, the application programmer specifies where the state of the object will be saved by choosing among the existing implementation of storage devices. The operations defined for persistent objects allow the application programmer to save or restore the state of the object at any time.

The framework does not rely on any kind of special programming language features. It only uses basic object-oriented programming techniques and is therefore implementable in any object-oriented programming language. Most of its structure is based on well-known design patterns. The remainder of this paper documents the construction of the framework.

3 Applying design patterns successively

3.1 Classification of storage types

At some point, persistent objects must save their state to some kind of storage, so that it can be retrieved again in a later execution. The term storage is used in a wider sense here. Sending the state of the object over a network and storing it in the memory of some other computer also makes sense, as long as the data survives program termination.

Nevertheless, all storage types do not have the same properties, and therefore must not all provide the same set of operations. An abstract class hierarchy is the most natural way to represent the structure of such storage types. A concrete implementation of a storage type must derive from one of the storage classes and implement the required operations. We propose to classify the different storage types in the way presented in Fig. 1.

The `Storage` class represents the interface common to all storage types. The operations `Read` and `Write` represent the operations that allow the user to read and write data from and to the storage device. What kind of value types they must support is discussed in more detail in the following subsection.

The storage hierarchy is split into 'volatile' storage and 'nonvolatile' storage. Data stored in volatile storage will not survive program termination. An example of a volatile storage is conventional computer memory. Once an application terminates, its memory is usually freed by the operating system, and therefore any data still remaining in it is lost. Data stored in nonvolatile storage on the other hand remains on the storage device even when a program terminates. Databases, disc storage, or even remote memory are common examples of nonvolatile storage. Since the data is not lost when the program terminates,

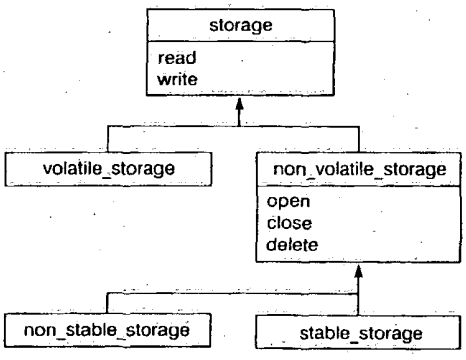

Fig. 1 `Storage` *class diagram*

additional housekeeping operations are needed to establish connections between the object and the actual storage device, to cut off existing connections, and to delete data that will not be used anymore. These operations are `Open`, `Close` and `Delete`.

Persistence can also be implemented in a stronger form to support fault tolerance. Tolerating software design faults (bugs), for instance by using the recovery block scheme [12], or tolerating faults of the underlying hardware, for instance by using checkpoints [13], requires some form of reliable storage.

This is why among the different nonvolatile storage devices we distinguish 'stable' and 'nonstable' devices. Data written to nonstable storage may get corrupted, if the system fails in some way, for instance by crashing during the write operation. Stable storage ensures that stored data will never be corrupted, even in the presence of application crashes and other failures.

The kind of storage to be used for saving application data depends heavily on the application requirements. To help the programmer choose the appropriate storage, every storage implementation must carefully document all interesting properties of the storage device. Properties such as throughput, average access time, capacity of the storage media and particularities of usage (for instance write-once devices such as CD-R burners) are of interest. Nonvolatile storage implementations must document the assumptions under which the storage is considered non-volatile, stable storage implementations must declare under what conditions they can be considered reliable. The application programmer, on the other hand, must examine the execution environment of the application, identify fault assumptions, and then, based on the information provided with each storage implementation, select the most suitable one.

3.2 Stable storage implementations

Stable storage has been first introduced in [14]. The paper describes how conventional disc storage that shows imperfections such as bad writes and decay can be transformed into stable storage, an ideal disc storage with no failures, using a technique called mirroring. When using this technique, data is stored twice on the disc (often two different physical discs are used to store the two copies of the data to increase reliability even more). If a crash occurs during the write operation of the first copy, the previously valid state can still be retrieved using the second copy. If the crash happens during the write operation of the second copy, the new state has already been saved in the first copy. When the system restarts after a crash failure, the state stored in the first copy must be duplicated and saved over the second copy.

Using this mirroring technique, any nonvolatile, nonstable storage can be transformed into stable storage. It is therefore possible to write an implementation of the mirroring algorithm that is independent of the actual nonvolatile storage that will effectively be used to store the data [15]. To achieve this decoupling, the 'strategy' design pattern described in [16] has been used. The 'strategy' design pattern has three types of participants: the 'strategy', the 'concrete strategy' and the 'context'.

The 'strategy', in our case the nonvolatile, nonstable storage class, declares the common interface to all concrete strategies. The 'context', in our case the mirroring class, uses this interface to make calls to a storage implementation defined by a 'concrete strategy'. This could be for instance a file storage class that implements storage based

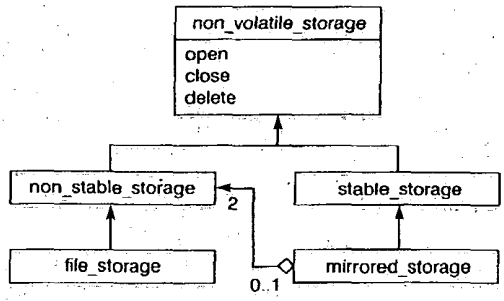

Fig. 2 *Stable storage using mirroring*

on the local file system. The structure of the collaboration is shown in Fig. 2.

At instantiation time, two nonvolatile storage objects must be passed as a parameter to the constructor of the mirroring class. That way, a variety of stable storage can be created reusing concrete implementations of non-volatile storage. What kind of nonvolatile storage will be chosen depends on the needs of the application.

The mirroring technique is not the only one that can be used to create stable storage. Database systems, for instance, have their own mechanism to guarantee atomic updates of data. Typically this is done by structuring updates of data as transactions. A transaction can be committed, in which case the updates will be made permanent, or aborted, in which case the system remains unchanged. If any kind of failure occurs during the transaction, the data also remains unchanged. It is possible to write a concrete stable storage class that provides a bridge between an object-oriented programming language and a database.

Yet another form of providing stable storage is replication. The state of a persistent object can be broadcasted over the network and stored, for instance, in remote memory. Although memory is usually considered volatile, it nevertheless is nonvolatile from the application point of view, since data stored in remote memory survives program termination. The group of replicas as a whole can be considered stable. As long as at least one of the remote machines remains accessible, the data can always be retrieved on a later execution.

Just as in the mirroring example, the replicated solution can be implemented in a generic way using the 'strategy' design pattern. This time, the relation between the context and the strategy is one-to-many as depicted in Fig. 3. The replication class implements broadcasting of data to remote memory and replica management algorithms that handle failures of replicas during program execution.

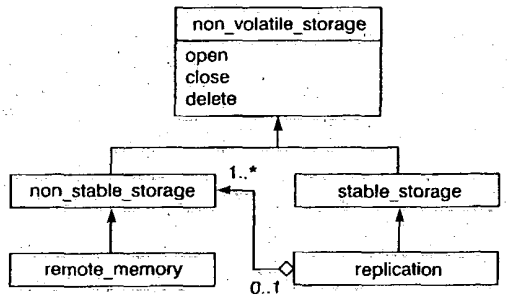

Fig. 3 *Stable storage using replication*

3.3 Object serialisation

When storing the state of a persistent object on some kind of storage device, the data must first be transformed from its representation in memory into some form that can be stored by the device. Most of the time the most convenient form will be a flat stream of bytes, e.g. for storing data in files or sending data though network transport buffers. Interfaces to ODBMs can be more elaborate.

The 'serialiser' design pattern described in [17] is an ideal solution for this kind of problem. It provides a mechanism to efficiently stream objects into data structures of any form, as well as create objects from such data structures. The participants of the 'serialiser' pattern are the 'reader/writer', the 'concrete reader/concrete writer', the 'serialisable' interface, 'concrete elements' that implement the 'serialisable' interface and different 'backends'. The structure of the 'serialiser' pattern is shown in Fig. 4.

The 'reader' and 'writer' parts declare protocols for reading and writing objects. These protocols consist of read, respectively write operations for every value type, including composite types, array types and object references. The 'reader' and 'writer' hide the 'backend' and the external representation format from the serialisable objects. 'Concrete reader' and 'concrete writer' implement the 'reader' and 'writer' protocols for a particular backend and external representation format.

The 'serialisable' interface defines operations that accept a 'reader' for reading and a 'writer' for writing. It also provides a `Create` operation that takes a class identifier as an argument and creates an instance of the denoted class. 'Concrete element' is an object implementing the 'serialisable' interface, making it possible to read and write its attributes to a 'concrete reader/concrete writer'.

The 'backend' is a particular backend, and corresponds to our storage class shown in the previous subsection. A 'concrete reader/concrete writer' reads from/writes to its backend using a backend specific interface. Relational database front-ends, flat files or network buffers are examples of concrete backends.

When invoked by a client, the 'reader/writer' hands itself over to the serialisable object. The serialisable object makes use of its protocol to read its attributes from a storage device or write its attributes to a storage device by calling the read/write operations provided by the 'reader/writer'. For certain value types such as composite types, the 'reader/writer' might call back to the serialisable object or forward the call to other objects that implement the 'serialisable' interface. This results in a recursive back-and-forth interplay between the two parties.

The larger the set of supported value types of the 'reader/writer' interface is, the more type information can be used by the 'concrete reader/concrete writer' to efficiently store the data on the backend. However, some backends support only a small set of value types. Flat files, for instance, only support byte transfer. For these kinds of backends the 'concrete reader/concrete writer' must contain implementation code that maps the `read/write` e operations of unsupported value types to the ones that are supported.

The big advantage of the 'serialiser' pattern is that the application class itself has no knowledge about the external representation format which is used to represent its instances. If this were not the case, introducing a new representation format or changing an old one would require to change almost every class in the system.

In some object-oriented programming languages, such a serialisation mechanism is already provided, which means that the `readFrom/writeTo` operations defined in the 'serialisable' interface have predefined implementations for all value types of the programming language that are not covered by the 'reader/writer' interface. The Java Serialisation package [18] or Ada streams (13.13 of [8]) are examples of such predefined language support. If no language support is available, the `readFrom/writeTo` operations of the 'serialisable' interface must be implemented for every 'concrete element'.

3.4 Creation of persistent objects

When creating an instance of a persistent object, the user must be able to specify on what kind of storage he wants the state of the object to be saved. The object can then create an instance of the corresponding storage class and thus establish a connection to the storage device.

The information needed to create an instance of a concrete storage class is device dependent. To create a new file, a user must typically provide a file name that follows certain conventions, and maybe also a path that specifies in which directory the file should be created. To access remote memory, an IP number or machine name must be provided. To solve this problem, a parallel hierarchy of storage parameters has been introduced. It has the same structure as the storage hierarchy (see Fig. 5). This allows each storage class to define it's own storage parameter type containing all the information it needs to uniquely identify data stored on the device.

At the same time, the storage parameter class allows a user to create instances of storage classes. This is done using the well-known 'factory method' pattern described in [16]. The participants of this design pattern are the 'product', the 'concrete product', the 'creator' and the 'concrete creator'.

The 'product' and 'concrete product' are in our case the storage class and its descendants, as they define and implement the interface of the objects the factory will create.

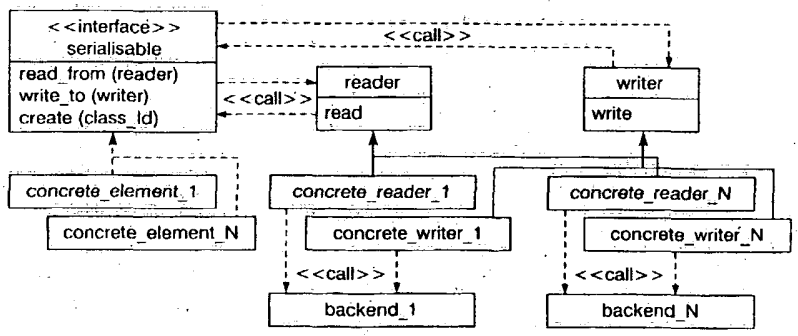

Fig. 4 *Serialiser pattern*

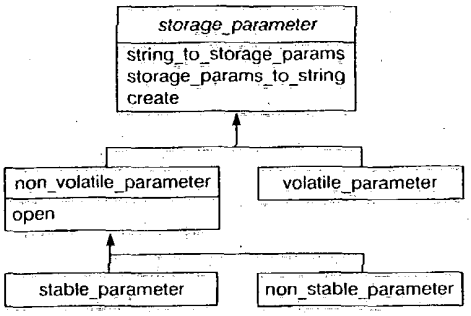

Fig. 5 `Storage_Parameter` *class hierarchy*

The 'creator' is the storage parameter class, for it declares the abstract factory method `Create`. A 'concrete creator', in our case a concrete storage parameter class, must provide an implementation for this method: the corresponding creator function of the storage class must be called, passing as a parameter the information stored inside the concrete storage parameter instance. Nonvolatile storage needs a second creator function, `Open`, that will instantiate the nonvolatile class without creating a new storage on the device. Instead, a connection between already existing data and the storage object will be established.

The `Create` and `Open` operations define the connection between the two parallel class hierarchies.

3.5 Identification of persistent objects

Since the state of a persistent object survives program termination there must be a unique way to identify a persistent object that remains valid during several executions of the same program. The storage parameter that has been introduced in the previous subsection uniquely identifies a location on the storage device, and can therefore also be used as a means for object identification.

As mentioned, storage parameters are device dependent. It can sometimes, however, be convenient for a user to treat persistent objects in a uniform way. An object name in the form of a string has proven to be an elegant solution for uniform object identification [6]. The two functions `Storage_Params_To_String` and `String_To_ Storage_Params` provide a mapping between the two identification means.

3.6 Storage management

Once a persistent object has been created and its state saved to a nonvolatile storage, the data will theoretically remain on the storage forever. The only way to remove the data and free the associated storage space is to explicitly delete the object. This situation can lead to permanent storage leaks, if the user forgets to store the parameters that allow him to identify the object on subsequent application runs.

Systems providing orthogonal persistence deal with this problem by periodically performing persistent garbage collection. Unreachable objects, i.e. objects that can not be designated by an application programmer anymore, can be safely deleted. Since the proposed framework does not assume any underlying services from the run-time, it does not have control over heap management. It is our intention to allow application programmers to access persistent objects just as other objects, so additional indirection can not be used to implement a controlled heap.

However, a simple solution to prevent permanent storage leaks is to provide some sort of reliable persistent directory. The parameters of every persistent object created so far are automatically stored in it. At any time, the user can consult the list of existing persistent objects to determine which of them he still needs and which of them he wants to delete.

Since the objects persist, the state of the directory should also survive program termination. The directory itself therefore is just another persistent object. When writing the state of the directory to the storage, the storage parameters of all persistent objects that have been created in the system must be written to the storage. It is therefore important that the storage parameter class also implements the 'serialisable' interface.

The directory must be reliable. Even a crash during the update of the directory should not corrupt the data. This can be achieved by storing the directory on stable storage. But this is not enough. Storage leaks can occur if a crash occurs after a new persistent object has been created, but before the creation has been registered in the directory. To prevent this problem, the creation and deletion of objects and the updating of the directory to reflect the change must be executed atomically.

4 Putting everything together

With the previous solutions in mind, we can now put together the overall system. Its structure is shown in Fig. 6. For simplicity, the 'reader' and 'writer' parts of the 'serialiser' pattern are shown as one class.

Before using any part of the framework, the user must initialise the persistence support. He has to choose where to store the persistent directory by instantiating the appropriate storage parameters and passing them to the `Initialise` operation of the persistence support class. To make the directory reliable, a stable storage must be used. A good idea is to hard-code the storage parameters of the persistent directory in the application code, for on subsequent runs the same parameters must be used again. During the `Initialise` operation, the persistent support class will try and restore a previously valid directory, or, if this fails, create a new, empty directory instead. Once the initialisation has been performed, the user can create, restore, save and delete persistent objects. The root class of the framework is the persistent object class. It implements the 'serialisable' interface described in the previous Section and the operations `Create`, `Restore`, `Save` and `Delete`.

The registered object class derives from the persistent object class. It is responsible for registering any new persistent object instances with the persistent directory. To define a new persistent object, the user must derive from the registered object class and add any application dependent state using new class attributes. He must also implement the `readFrom` and `writeTo` operations of the 'serialisable' interface, if the underlying programming language does not provided them automatically.

The user-defined class must also provide two constructors: `Create`, to create a new instance of a persistent object, and `Restore`, to restore the state of an already existing persistent object. They must perform initialisation of the application dependent object state, if needed, and then up-call the corresponding constructor of the registered object class, which performs the necessary operation for registering the object with the persistent directory.

The `Create` operation of the registered object class will call the `Create_Object` operation of the persistent

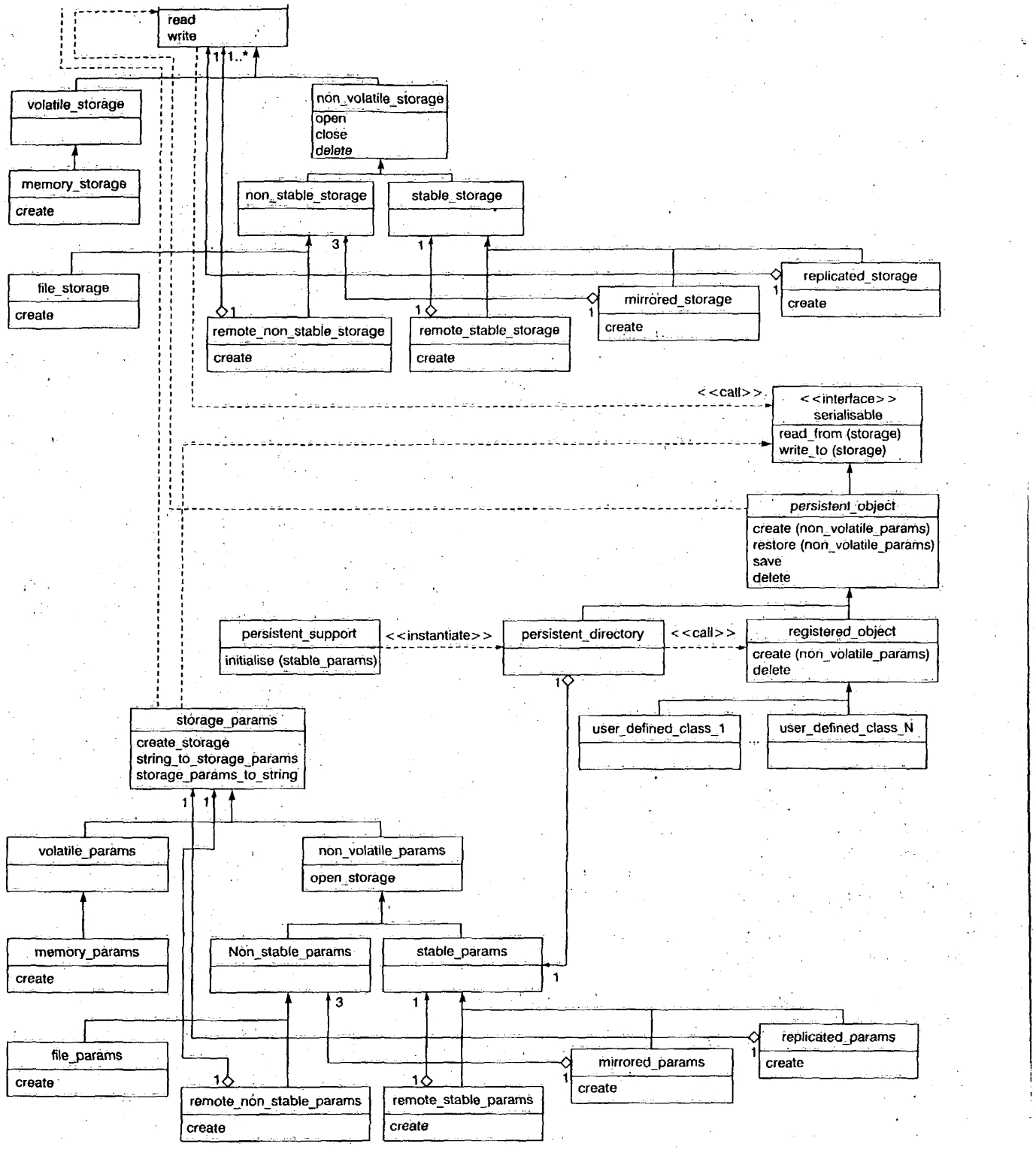

Fig. 6 *Framework overview*

directory. Inside `Create_Object`, the operation `Create` of the persistent class is called. This will actually create the object on the specified store as explained below. At the same time, the storage parameters of the newly created class are stored inside the persistent directory. As mentioned, this must be done atomically. A simple way of implementing this is to use a simple form of logging.

The implementation of the constructors `Create` and `Restore` in the persistent object class will create an instance of the storage type identified by the storage parameter using the `Create` or `Open` factory methods. If `Restore` has been called, the state of the object is read from the storage using the operations of the 'serialisable' interface.

This is again an application of the 'strategy' design pattern. The persistent object, the 'context', is configured with a storage, the 'strategy', at instantiation time. The user chooses the storage medium for his object by passing the corresponding storage patterns to the `Create` or `Restore` constructors. Once this association has been

set up, the user can write the state of the persistent object to the associated storage using the Save operation. The implementation then writes an object identifier and successively all object attributes to the associated storage device.

5 Experimental implementation

An experimental implementation of the framework [19] has been realised using the object-oriented programming language Ada 95 [8].

5.1 Storage hierarchy

The fact that Ada 95 supports streaming of objects has simplified the implementation, but also narrowed down the read/write operations of the storage type to support byte reads and writes only. This fact is reflected in the specification of the abstract Storage_Type. There is only one pair of read/write operations, and it operates on stream element arrays (arrays of bytes):

```
with Ada.Streams; use Ada.Streams;
package Storage_Types is
  type Storage_Type (<>) is abstract
    tagged limited private;
  type Storage_Ref is access all
  Storage_Type'Class;
  procedure Read
    (Storage: in out Storage_Type;
    Item : out Stream_Element_Array;
    Last : out Stream_Element_Offset) is
    abstract;
  procedure Write
    (Storage: in out Storage_Type;
    Item : in Stream_Element_Array) is
    abstract;
private
  ...
end Storage_Types;
```

A persistence support programmer writing a new interface for a storage device must derive from this class (or more precisely from a subclass such as Non_Stable_Storage_Type, whose properties correspond to the properties of the device) and implement the Read and Write operations.

5.2 Declaring a persistent type

To use the framework, the programmer must first declare his own persistent type by deriving from the registered object class, here called Registered_Object_Type, adding additional attributes that contain the application data. The following example shows how to declare a persistent integer type:

```
with Persistent_Objects.Registered;
use Persistent_Objects.Registered;
package Persistent_Integers is
  type Persistent_Integer_Type is new
    Registered_Object_Type with record
    Value : Integer;
  end record;
  type Persistent_Integer_Ref is access
  all
    Persistent_Integer_Type'Class;
  function Create
    (Storage_Params : Non_Volatile_
    Params_Type'Class)
    return Persistent_Integer_Ref;
end Persistent_Integers;
```

The implementation of the constructor must allocate the new object, performs initialisation and then calls up the constructor of the registered object class as shown in the following code:

```
package body Persistent_Integers is
  function Create
    (Storage_Params : Non_Volatile_
    Params_Type'Class)
    return Persistent_Integer_Ref is
    Result : Persistent_Integer_Ref :=
    new Persistent_Integer_Type;
begin
  Create (Registered_Object_Ref
  (Result), Storage_Params);
  return Result;
end Persistent_Integers;
```

Per default, Ada streaming writes the entire object, i.e. the values of all its components to the corresponding stream. The application programmer can modify this standard behaviour, for instance, to make only parts of an object's state persistent. If the application program evolves over time and the static structure of a persistent class has been changed, then default streaming must be changed in order to still be able to read previously saved versions of the class. The following Ada code shows how this can be done by replacing, for instance, the default read procedure with the procedure My_Read:

```
package Persistent_Integers is
  ...
private
  procedure My_Read
    (Stream : access Ada.Streams.Root_
    Stream_Type'Class;
    Item : out Persistent_Integer_Type);
  for Persistent_Integer_Type'Read use
    My_Read;
end Persistent_Integers;
```

5.3 Using the framework

Based on the declarations shown in the previous subsection, instances of the persistent integer class can now be created and used.

```
- include the necessary files
with File_Storage_Params;
with Persistent_Integers; use
Persistent_Integers;
- Create a new persistent integer and save
- it's contents
declare
  I : Persistent_Integer_Ref :=
    Create (File_Storage_Params.String_
    To_Storage_Params (''foo''));
begin
  I.Value := 123;
  Save (I.all);
 nd;
- Restore the contents of the previously
- created persistent integer
declare
  I : Persistent_Integer_Ref :=
  Persistent_Integer_Ref
```

```
  String_To_Storage_Params
    (''foo'')));
begin
  -- work with I
end;
```

When creating the persistent integer, the user chooses on which storage the state shall be stored by calling the `String_To_Storage_Params` function of the chosen storage class. In the example above, the persistent integer is stored in a file, since `String_To_Storage_Params` of the `File_Storage_Params` class is called.

6 Related work

Existing persistent systems can be categorised into three main categories: persistent programming languages, operating systems with support for persistence, and services providing persistence.

6.1 Persistent programming languages

The first language providing orthogonal persistence, PS-Algol [4], was conceived to add persistence to an existing language with minimal perturbation to its initial semantics and implementation. There are persistent versions of functional programming languages such as Persistent Poly and Poly ML [5].

There has also been work on adding orthogonal persistence to widely used programming languages. Probably the most interesting project nowadays is PJama [6, 20], a project that aims at providing orthogonal persistence to the Java [11] programming language without modifying the language itself. Roots of persistence have been defined, where individual objects can be registered during run-time. All objects reachable from a persistent root are made persistent (persistence by reachability). This is achieved by modifying the Java Virtual Machine.

6.2 Operating systems with persistence support

There are numerous examples of operating systems that support persistence in some form, for instance MONADS [21], Clouds [22] and Grasshopper [23].

Grasshopper is an operating system explicitly designed to support orthogonal persistence. It provides three abstractions: 'loci', 'capabilities' and 'containers', all of which are inherently persistent. Loci abstract over execution. Containers abstract over storage. They are conceptually very large address spaces capable of holding persistent data. Data stored inside a container are referenced directly using an address relative to the start of the container. Containers can exist independently, i.e. unlike conventional operating systems, address spaces are not tied to processes. Finally, capabilities control the access to containers and loci.

6.3 Services providing persistence

The CORBA Persistent Object Service [24] is a standardised CORBA service that allows CORBA Objects to make all or part of their state persistent. Whether or not the client of such a persistent object is aware of the persistent state is a choice the object has. By supporting special interfaces, and describing the persistent data using an interface definition language, a persistent object can delegate the management of its persistent state to other objects.

The persistent data finally get stored in so-called data-stores. This is the CORBA way to abstract the real storage devices, such as file systems or databases. Each persistent object can dynamically be connected to a data-store. From then on it is possible to save and restore the persistent state.

The PerDiS project [25] takes a very different approach to persistence. They address the issue of providing support for distributed collaborative engineering applications such as CAD programs, where large volumes of fine-grain, complex objects must be shared across wide-area networks. They present the user with some form of a persistent, distributed memory. The application accesses this memory transactionally. The memory is divided into clusters containing objects. Named roots provide the entry points. Objects are connected by pointers. Reachable objects are stored persistently in clusters on disc; unreachable objects are garbage-collected automatically.

Transactional systems often incorporate a persistence support for guaranteeing the durability property. Several object-oriented transactional systems use class libraries to declare transactional objects and to manipulate them. For example, Arjuna [26], one of the best known object-oriented transactional systems, is delivered as a set of C++ classes. It provides two types of object stores: persistent (the stable storage is implemented as a set of UNIX files) and volatile (implemented in the main memory). These systems usually hide all details of the ways they implement persistence and do not offer programmers any object-oriented extendable approaches to manipulate the stable storage (e.g. to include a new storage class to support a new media, or, to change the storage dynamically). They incorporate object storage control into the transactional control.

7 Conclusions

We have presented the construction of an object-oriented framework providing persistence support for object-oriented programming languages without modifying either the language itself or its run-time support. It relies on basic object-oriented programming techniques only, and can therefore be implemented and used in any object-oriented programming language. The chief advantage of this approach is that the proposed framework can be applied in any settings/platforms with standard compilers and run-time. We perfectly well realise, however, that the presented approach does not meet all requirements of orthogonal persistence.

The advantage of using object-oriented programming in this context is obvious. Class hierarchies have allowed us to clearly classify various storage devices and show the relations between them. Abstract classes provide the interface for various types of storage. The structure of the framework is based on well-known design patterns. The advantages of using design patterns in this context are substantial, since they enhance modularity and flexibility of object-oriented programs, and help programmers to achieve their goals in a disciplined and error-free fashion. Design patterns represent solutions to specific problems that have proven to be successful. In addition, people familiar with the design patterns used in the framework will be able to understand the structure of the framework faster. The design patterns used and their advantages are summarised as follows:

- The 'serialiser' pattern makes it easy to add new data representation formats for objects whose state has to be stored in new storage devices by introducing a new

'reader/writer' pair. It also moves the knowledge about the external data representation format out of the persistent object itself.

- Encapsulating storage devices in separate 'strategy' classes lets the user change and replace particular storage implementations, or even extend the storage device hierarchy without modifying the persistent object class. It also promotes reuse when implementing stable storage on top of nonvolatile storage.
- Using the 'factory method' pattern combined with a parallel class hierarchy representing storage parameters makes it possible to provide a uniform way for persistent objects to create storage devices. At the same time, the storage parameter classes provide unique identification of persistent objects.

To prove the concept, the framework has been implemented for the standard object-oriented programming language Ada 95. Simply by using derivation, a programmer can declare all instances of a class to be persistent. The implementation uses the standard streaming facilities of Ada to serialise the state of persistent objects. If needed, the default serialisation can be replaced with a new one by the application programmer.

The framework presented has been used as persistence support in Optima, a framework providing support for transactions in programming languages [27, 28], which has been implemented in Ada 95. This implementation has been used in the development of a distributed, transaction-based auction system [29].

8 Acknowledgments

Jörg Kienzle has been partially supported by the Swiss National Science Foundation project FN 2000-057187.99/1. Alexander Romanovsky has been partially supported by the EC IST RTD Project on Dependable Systems of Systems (DSoS).

9 Refer nces

1 BOOCH, G.: 'Object-oriented design with applications' (Benjamin/Cummings, Redwood City, CA, USA, 1991)
2 ATKINSON, M.P., and MORRISON, R.: 'Orthogonally persistent object systems', *VLDB J.*, 1995, **4**, (3), pp. 319–401
3 ATKINSON, M.P., BAILEY, P.J., CHISHOLM, K.J., COCKSHOTT, W.P., and MORRISON, R.: 'An approach to persistent programming', *Comput. J.*, 1983, **26**, (4), pp. 360–365
4 ATKINSON, M.P., CHISHOLM, K.J., and COCKSHOTT, W.P.: 'PS-Algol: an algol with a persistent heap', *SIGPLAN Not.*, 1981, **17**, (7), pp. 24–31
5 MATTHEWS, D.C.J.: 'A persistent storage system for Poly and ML'. Technical report TR-102, Computer Laboratory, University of Cambridge, January 1987
6 ATKINSON, M.P., DAYNÈS, L., JORDAN, M.J., PRINTEZIS, T., and SPENCE, S.: 'An orthogonally persistent Java', *SIGMOD Rec.*, 1996, **25**, (4), pp. 68–75
7 OUDSHOORN, M.J., and CRAWLEY, S.C.: 'Beyond Ada 95: the addition of persistence and its consequences'. Proceedings of Reliable Software Technologies – Ada-Europe '96, Montreux, Switzerland, 10–14 June 1996, pp. 342–356; Lect. Notes Comput. Sci., vol. 1088 (Springer Verlag, 1996)
8 ISO: International Standard ISO/IEC 8652: 1995(E): Ada Reference Manual, ISO, Geneva, 1995; Lect. Notes Comput. Sci., vol. 1246 (Springer Verlag, 1997)
9 ATKINSON, M.P., JORDAN, M.J., DAYNÈS, L., and SPENCE, S.: 'Design issues for persistent Java: a type-safe, object-oriented, orthogonally persistent system'. Proceedings of the 6th International Workshop on Persistent object systems, Cape May, NJ, USA, May 1996
10 EPPINGER, J.L., MUMMERT, L.B., and SPECTOR, A.Z.: 'Camelot and Avalon – a distributed transaction facility' (Morgan Kaufmann, San Mateo, CA, 1991)
11 GOSLING, J., JOY, B., and STEELE, G.L.: 'The Java language specification' (Addison Wesley, Reading, MA, USA, 1996)
12 RANDELL, B.: 'System structure for software fault tolerance', *IEEE Trans. Softw. Eng.*, 1975, **1**, (2), pp. 220–232
13 LEE, P.A., and ANDERSON, T.: 'Fault tolerance – principles and practice' in 'Dependable computing and fault-tolerant systems' (Springer Verlag, 1990, 2nd edn.)
14 LAMPSON, B.W., and STURGIS, H.E.: 'Crash recovery in a distributed data storage system'. Technical report, Xerox Research, Palo Alto, CA, June 1979
15 CARON, X., KIENZLE, J., and STROHMEIER, A.: 'Object-oriented stable storage based on mirroring'. Proceedings of Reliable Software Technologies – Ada-Europe'2001, Leuven, Belgium, 14–18 May 2001, pp. 278–289; Lect. Notes Comput. Sci., vol. 2043 (Springer Verlag, 2001)
16 GAMMA, E., HELM, R., JOHNSON, R., and VLISSIDES, J.: 'Design patterns' (Addison Wesley, Reading, MA, USA, 1995)
17 RIEHLE, D., SIBERSKI, W., BÄUMER, D., MEGERT, D., and ZÜLLIGHOVEN, H.: 'Serializer' in 'Pattern languages of program design, vol. 3' (Addison-Wesley, Reading, MA, USA, 1998), pp. 293–312
18 Sun Microsystems: 'Java object serialization specification', November 1998
19 KIENZLE, J., and ROMANOVSKY, A.: 'On persistent and reliable streaming in Ada'. Proceedings of Reliable Software Technologies – Ada-Europe'2000, Potsdam, Germany, 26–30 June 2000, pp. 82–95; Lect. Notes Comput. Sci., vol. 1845, 2000
20 DAYNÈS, L.: 'Implementation of automated fine-granularity locking in a persistent programming language', *Softw. — Pract. Exp.*, 2000, **30**, (4), pp. 325–361
21 ROSENBERG, J.: 'The MONADS architecture: a layered view'. Fourth International Workshop on Persistent object systems, Martha's Vineyard, MA, USA, September 1990, pp. 215–225
22 DASGUPTA, P., RICHARD, J., LEBLANC, J., AHAMAD, M., and RAMACHANDRAN, U.: 'The clouds distributed operating system', *Computer*, 1991, **24**, (11), pp. 34–44
23 DEARLE, A., DI BONA, R., FARROW, J., KENSKENS, F., LINDSTROM, A., ROSENBERG, J., and VAUGHAN, F.: 'Grasshopper: an orthogonally persistent operating system', *Comput. Syst.*, 1994, **7**, (3), pp. 289–312
24 Object Management Group, Inc.: 'Externalization service specification', December 1998
25 FERREIRA, P., SHAPIRO, M., BLONDEL, X., FAMBON, O., GARCIA, J., KLOOSTERMAN, S., RICHER, N., ROBERTS, M., SANDAKLY, F., COULOURIS, G., DOLLIMORE, J., GUEDES, P., HAGIMONT, D., and KRAKOWIAK, S.: 'PerDiS: design, implementation, and use of a Persistent Distributed Store'. Technical report, QMW TR 752, CSTB ILC/98-1392, INRIA RR 3525, INESC RT/5/98, October 1998
26 PARRINGTON, G.D., SHRIVASTAVA, S.K., WHEATER, S.M., and LITTLE, M.C.: 'The design and implementation of Arjuna', *Comput. Syst.*, 1995, **8**, (3), pp. 253–306
27 KIENZLE, J., JIMÉNEZ-PERIS, R., ROMANOVSKY, A., and PATIÑO-MARTINEZ, M.: 'Transaction support for Ada'. Proceedings of Reliable Software Technologies – Ada-Europe'2001, Leuven, Belgium, 14–18 May 2001, pp. 290–304; Lect. Notes Comput. Sci., vol. 2043 (Springer Verlag, 2001)
28 KIENZLE, J.: 'Open multithreaded transactions: a transaction model for concurrent object-oriented programming'. PhD thesis 2393, Swiss Federal Institute of Technology, Lausanne, Switzerland, April 2001
29 KIENZLE, J., ROMANOVSKY, A., and STROHMEIER, A.: 'Auction system design using open multithreaded transactions'. In Proceedings of the 7th International Workshop on Object-oriented real-time dependable systems, San Diego, CA, USA, 7–9 January 2002 (IEEE Computer Society Press, 2002), pp. 95–104